AD-A053 264 BROWN UNIV PROVIDENCE R I DIV OF APPLIED MATHEMATICS F/G 5/10
ABDUCTION MACHINES THAT LEARN SYNTACTIC PATTERNS,(U)
1978 U GRENANDER N00014-75-C-0461
UNCLASSIFIED NL

| OF |
AD
A053264

END
DATE
FILMED
6-78
DDC

(15) Contract N00014-75-C-0461
Office of Naval Research and Brown University

(11)

Code 23
05

(11) 1978

(12) 28 p.

(6) Abduction machines that learn syntactic patterns,

by

(10) Ulf Grenander
L. Herbert Ballou University Professor

Division of Applied Mathematics

Brown University

Report No. 25 in the Pattern Analysis Series.

065 300

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DDC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

1. To learn syntactic patterns.

The following study was started following a discussion with Leon Cooper, Walter Freiberger, and Henry Kucera. The topic had been language learning, how children learn a language, and to what degree it is possible to construct a model - an abduction machine - that can perform such tasks.

We singled out one part of language learning, namely to discover the syntax. This means that we leave aside the intricate problems preceding syntax learning, such as speech segmentation, learning and recognizing words, and semantic association.

Also we would of course try any abduction machine on formal languages rather than natural language. It seems natural to start with some simple classes of languages.

A question that arises immediately is whether our problem is solvable in the sense of logic. It is well known that some seemingly simple questions in formal languages are not solvable. To mention just one example: it is unsolvable if the languages produced by two context free grammars are identical.

However, this is not what we are really trying to find out. Indeed, if we could construct an abduction machine that performed its task in principle but required an enormous computing effort to do it, then we would not consider this a satisfactory solution. Also we would require that the functioning of the machine should be natural in the sense that its operation resemble the way that humans learn syntax, or at least does not seem completely artificial.

On the other hand, if we had an unsolvable problem at hand, but could produce a machine that produced a learning effect with reasonable computing effort and with results that approximate the true grammar, then we may be willing to accept it as a solution to our problem.

In other words, what we are after is not competence but performance. Some important consequence of this decision will turn up in later sections.

It is obvious that the amount of preprogramming of the neural net that is required for this task will depend upon how general is the class of grammars that we allow. For the present paper we shall assume that the grammars are at least context free and most of the time even finite state grammars.

The neural machines used will resemble the ones we have used in earlier work. They can still be considered as dynamical sytems in a wide sense, but where the system is not represented by ordinary differential equations but have the structure of automata.

We shall perform experiments that will help us understand how abduction machines should be constructed. For this purpose it will be convenient to have available a program that will help us set up the grammar easily and will generate sentences according to it.

The model used will be the one introduced in Grenander (1966) and studied in some later reports. With a given lexicon of words a,b,c,... we postulate context free rules of the form

(1.1) v → string

where v is one of the syntactic variables and string is made up of words and variables. The variables will be labeled by numbers 1,2,3,... . For a given variable v we assume a probability distribution over the rules that rewrite that symbol. We have shown elsewhere what properties those probability distributions must have in order that the probability measure induced over the set of all finite word-strings be a real one in the sense that this set have total probability one. It will be assumed that these properties hold for the grammars we define. We then speak of a syntax-controlled probability grammar.

A convenient vehicle for doing this is the APL program SETUP given in the Appendix. It interrogates the user about the size of the lexicon, what syntactic rules, what probabilities associated with the rules and so on. Then the function REWRITE will generate one grammatical sentence according to the model. Repeated calls of REWRITE will give us a sample from the language.

It will be instructive for the user to generate a sample and see how a human would try to discover the underlying grammar. It is not so easy, even for simple grammars. We now turn to the main question: construc a machine that does it.

2. A naive abduction machine.

How does someone learn a language without explicit instruction in its grammar? Do children group words into word classes, employ concepts like parts of speech, search for rules? Whether this is os or not it is at least true that this is the way grammarians have been operating from Panini and Thrax on.

Let us try to imitate this. To put it more formally let us consider two strings u and v of words. The strings need not themselves be grammatical sentences. If it is true for any pair x,y of strings that xuy and xvy are simultaneously grammatical or ungrammatical the original strings are said to be equivalent, written as uEQv. Obviously EQ is an equivalence relation and it divides the set of all finite strings into equivalence classes. This is an established approach in formal linguistics.

To simplify the discussion we shall assume that our language $\mathcal{L}$ is a finite state language, which is the same thing as to say that it is generated by a finite state grammar G, $\mathcal{L} = L(G)$. Then, after some simple changes all the syntactic rule take the form, either

(2.1) $\quad i \rightarrow aj$

where i and j are syntactic variables and a is some word from the lexicon, or

(2.2) $\quad i \rightarrow a$.

In the latter case one speaks of a terminating rule. (2.1) and (2.2) are rules of a right linear grammar.

Let us take as an example n_w = number of words = 3, so that LEXICON = {a,b,c} and n_v = number of variables = 3, so that i=1,2,3. The rules with their probabilities could be

$$
(2.3) \qquad \begin{cases} 1 \to a1 & .5 \\ 1 \to b2 & .5 \\ 2 \to b3 & .5 \\ 2 \to c & .5 \\ 3 \to a1 & .5 \\ 3 \to b & .5 \end{cases}
$$

Equivalent to this is the deterministic finite state automaton in Figure 2.1. The automaton starts in state 1 and follows the transitions in the wrong diagram according to the successive words in the input string. When it comes to the end of the string the sentence is accepted as grammatical if and only if

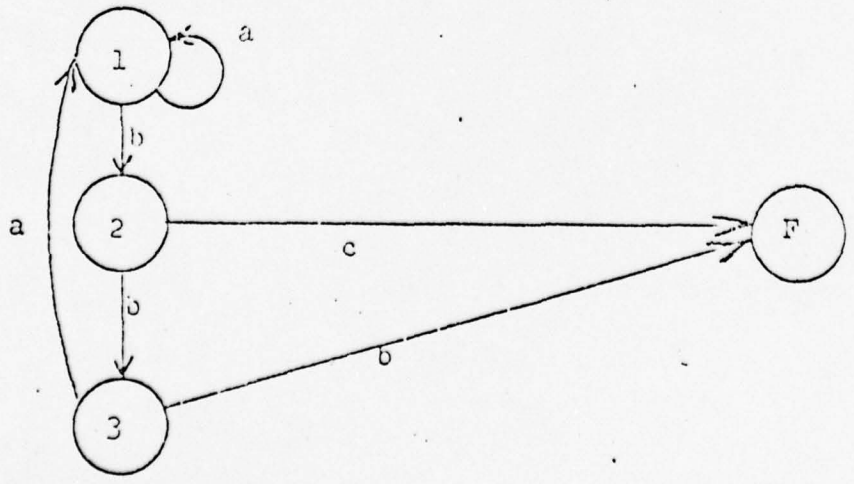


Figure 2.1

the machine is in the final state F. The function SETUP generated a sample beginning with

(2.4)

AABC
BBB
AABBAAABBAABC
BBABBABBB
BBABC
ABC
BC
BBB
ABBAAABC
BC
AAAAAABC
...

For a right linear grammar one need only consider uy and vy to define equivalence. If u = ab and v = b it is seen that with these two strings as a beginning the automaton in Figure 2.1 will be in state 2. Whatever follows it is clear that the result will be either grammatical or ungrammatical so that aEQab. They belong to the same equivalence class. On the other hand the string bb ends in stabe 3 so that bb belongs to another equivalence class. The string cb cannot be generated by the machine but it will be convenient to add a new state to take case of such strings and their equivalence class. Then it is seen that the states of the automaton and the associated syntactic variables correspond directly to the equivalence classes.

Consider now the reactions of the imagined child to its language environment. When it listens to presumably grammatical sentences it would look for equivalence classes. To represent them it is enough to select one string from each and short strings will be adequate unless the number of variables is quite large. Say that one considers the set of all strings of length at most equal to d, for depth, as candidates.

The maximum number of equivalence classes that could be found in this way is

$$(2.5) \quad n_c = n_w + n_w^2 + n_w^3 + \ldots + n_w^d = \frac{n_w^d - 1}{n_w - 1} n_w \ .$$

Note that n_c increases fast with d.

As sentences are being encountered the listerner tries the initial substrings up to depth d as candidates for the distinct equivalence classes. A sentence ux is compared with earlier encountered sentences of the form vx. If they can be found in memory it also looks for sentences uy and vy. If an instance is found in which uy is grammatical but vy not it is clear that u and v are not equivalent.

Create a matrix $n_c \times n_c$ and put a large negative number in the cell corresponding to the indices of u and v each time the above event occurred: this signifies that u and v belong to different classes. On the other hand if uy and vy are grammatical for some y increase the value in the cell one unit.

The latter is done by speaking. The sentence vy is spoken and if it is accepted by the environment as correct the cell value of the matrix is increased as mentioned.

As time goes on the partition will be finer, lack of equivalence will be established with certainty, while equivalence only is gradually increased. We now apply a threshold logic. If u and v have a cell value which is positive we treat them (temporarily) as equivalent. This will not necessarily be a true equivalence relation since it need not be transitive. We therefore have to extend it by forming chains by pairwise equivalent pairs of initial strings.

The matrix is initialized by putting all entries equal to a moderately large negative number: the tabula rasa hypothesis. The flow chart looks as in Figure 2.2.

The block SPEAK AND TEST involve a good deal of computing with comparisons and matching. Here we need a "teaching" program called ACCEPT, see Appendix, that will also be used in a more ambitious abduction machine.

This machine was coded and run with the following experiences. The machine worked in the sense that for very small number of variables, they and the corresponding rules were eventually discovered. In spite of this it was deemed a failure for three reasons.

1. Even in these extremely simple cases it was exasperatingly slow.

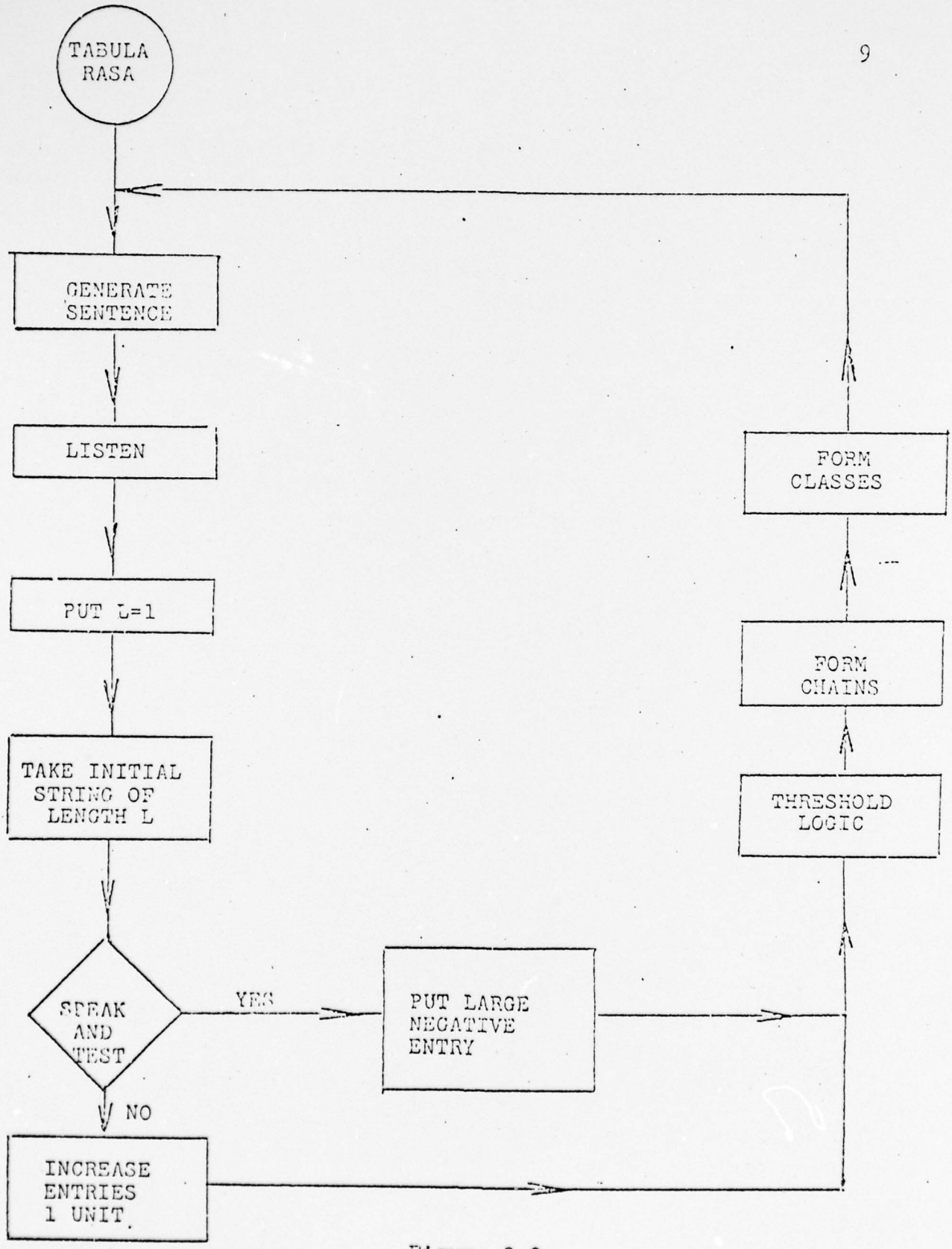

TABULA RASA
GENERATE SENTENCE
LISTEN
PUT L=1
TAKE INITIAL STRING OF LENGTH L
SPEAK AND TEST
YES
NO
PUT LARGE NEGATIVE ENTRY
INCREASE ENTRIES 1 UNIT
THRESHOLD LOGIC
FORM CHAINS
FORM CLASSES

Figure 2.2

2. The space requirement was growing at an unacceptable rate as each (new) sentence heard had to be stored in its memory. This is very unlikely in human learning, perhaps impossible because the storage volume required.

3. The search in SPEAK AND TEST and the following blocks in Figure 2.2 is too systematic. Whatever way humans may use to learn grammar this is not it.

We therefore decided to scrap this abduction machine and build a better one.

3. A smarter machine.

The idea to look for equivalence classes still seems promising, but not the way it was done. To visualize the neural network needed for the computing in Figure 2.2 look at the diagram in Figure 3.1.

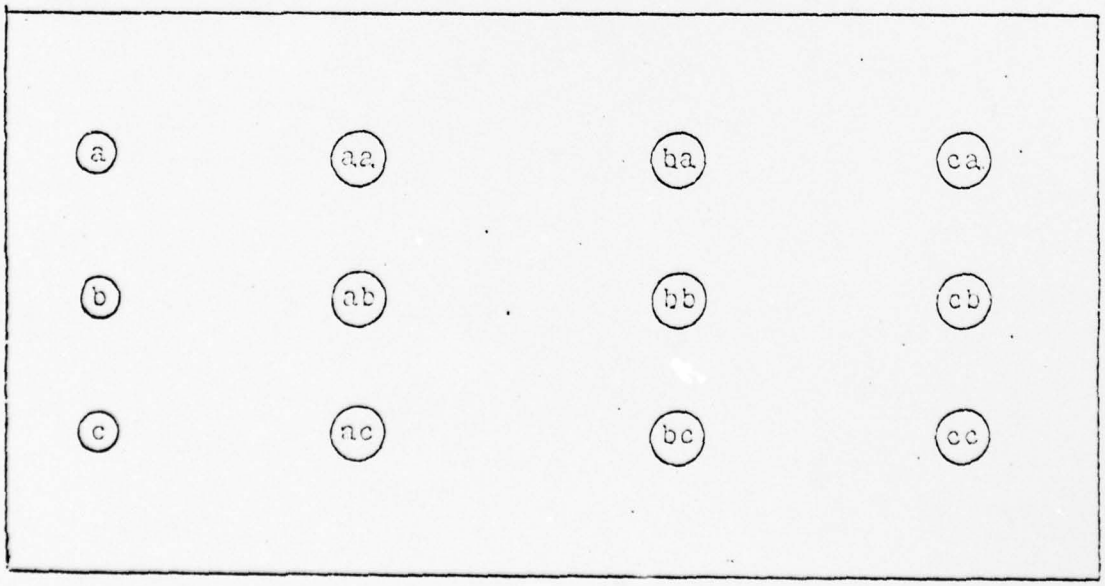


Figure 3.1

Here $n_w = 3$, $d = 2$, $n_c = 12$. We set up a connectivity matrix of size 12×12. We do not need more than the entries above the main diagonal which means $n_c(n_c-1)$ values, in the picture = 132. At TABULA RASA all these values will be moderately large negative numbers which will be updated following the results of SPEAK AND TEST. Already for slightly larger values for n_w and n_v this would lead to enormous storage requirement.

What is wrong in this machine is that it proceeds by aggregation: starting from lots of possible equivalence classes they are coalesced into fewer until we have arrived at the true number of classes.

Instead we should start from the opposite end. Assuming that there is only one equivalence class (the function TABULA in the Appendix) the machine listens to grammatical sentences. It takes initial substrings and determines its number and class (from the updated matrix CLASS). This class may contain several other substrings. One of them is selected at random (no systematic search) and it will be tested for equivalence with the first one. Depending upon the outcome of the comparison either a new variable (row in CLASS) is created, or a substring is moved from one row to another, or no action is taken. All of this is done by the program LINGUA which calls the comparison function TEST, see Appendix.

We now claim that this abduction machine is consistent in the following performance sense.

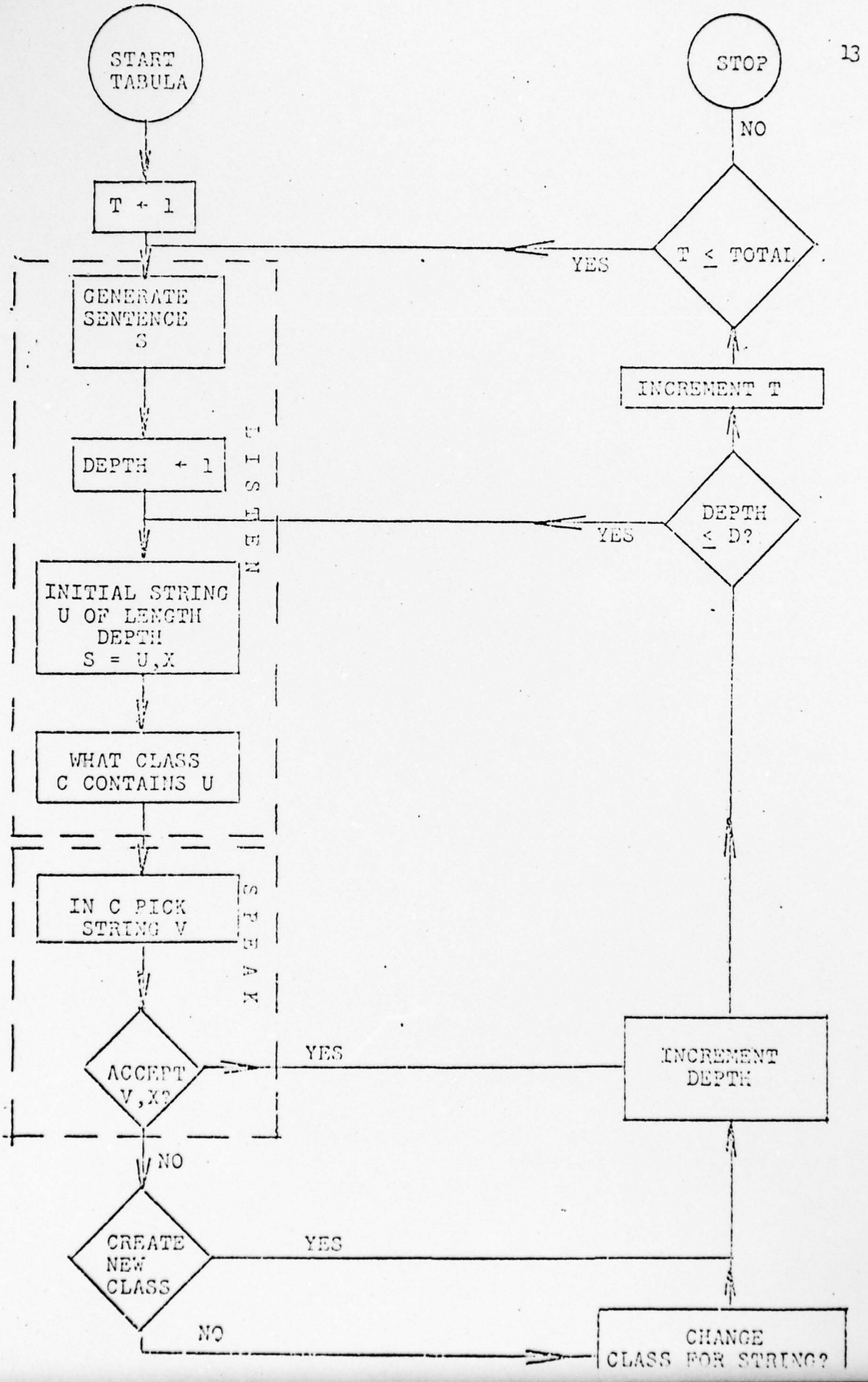
Theorem. Consider the algorithm described and represented by the flowchart in Figure 3.2. Assume that DEPTH d the depth of the true grammar. As the total number of iterations, TOTAL, tends to infinity learning of the grammar occurs with probability one: the number of classes and the classes themselves will converge to a limit such that it and the corresponding rules form a grammar equivalent with the true one.

Proof: Since the search is not systematic but (to some extent) random any convergence must be of a probabilistic kind.

Say now that for a large number of T a certain set of classes have been established. As new sentences come along two action can be taken: establish a new class or move an initial substring from one class to another.

The first action will occur if the sentence starts with a u still belonging to the first class set up in TABULA, but representing a real syntactic variable distinct from the initial symbol. If u has not yet been established there is a positive probability in each iteration T that it will be discovered. Hence with probability zero this will happen after a finite number of iterations.

The second action means that it is discovered that two strings u and v, that are put in the same class temporarily, will be detected to be non-equivalent but v is believed (for the moment) to be equivalent to some w belonging to an already established class. Then v will be moved to the class of w. In order that this should happen let us assume that all classes


START TABULA
T ← 1
GENERATE SENTENCE S
DEPTH ← 1
INITIAL STRING U OF LENGTH DEPTH S = U,X
WHAT CLASS C CONTAINS U
IN C PICK STRING V
ACCEPT V,X?
NO
CREATE NEW CLASS
NO
CHANGE CLASS FOR STRING?
YES
INCREMENT DEPTH
YES
DEPTH ≤ D?
YES
INCREMENT T
T ≤ TOTAL
YES
NO
STOP
TESTEN
SPEAK

have been created (we already know that this will happen after finite time). Due to the finite depth only a finite number of arrangements exist for the grouping of the initial strings. For each one there is a positive probability that an incorrect grouping shall be detected in one trial. Hence, as $T \to \infty$, all the equivalence classes will be correctly established after a finite number of iterations, again with probability one.

This concludes the proof of convergence but it should be noticed that the positive probabilities mentioned may be small which will result in a very low learning rate.

It is clear that this abduction machine has only a modest memory requirement compared to the earlier one. Old sentences need not be remembered, nor is it necessary to store the enormous matrix that represented the syntactic relationships in quantitive terms of strength of belief.

The learning mechanism also appears less artificial. It still has the LISTEN-SPEAK cycle but the spoken sentences are being corrected in a less systematic manner.

It should also be remarked that no claim is made that the original grammar is learnt in exactly the same form as it has been defined. The claim is that as it converges in the sense of weakly equivalent grammars.

As a matter of fact the search for equivalence classes will result in a limiting grammar with a minimum number of variables or classes. In this sense the abduction machine appeals to Occam's razor.

To test for speed of convergence the machine was exposed to various finite state languages. The speed of learning was much higher in general than for the naive machine. For example, the grammar (2.1), which took very long to learn before was now learnt almost immediately. For T=1 the variables B and BB were established, for T=2 the variables A, for 6=5 the variable BC:

```
      TABULA
      LINGUA 20
NEW VARIABLE B  CREATED AT SENTENCE NO. 1 = BBABBAABBB
NEW VARIABLE BB CREATED AT SENTENCE NO. 1 = BBABBAABBB
NEW VARIABLE A  CREATED AT SENTENCE NO. 2 = ABBB
NEW VARIABLE BC CREATED AT SENTENCE NO. 6 = BC
```

In other runs the performance of the abduction machine was similar, in no case requiring more than 10 iterations before convergence was established.

For other finite state grammars with many variables and rules the time it took was longer correspondingly but no exorbitant number of iterations was required in this series of tests.

I would like to report on one test, not because the grammar was more complicated, but because the result seemed surprising at first and led to some reflections on the notion of style that ought to be explored in more depth.

I had been looking at languages where the sentence is not just expressed as a linear string of words but takes the form of a colored picture. More precisely, the question was whether one

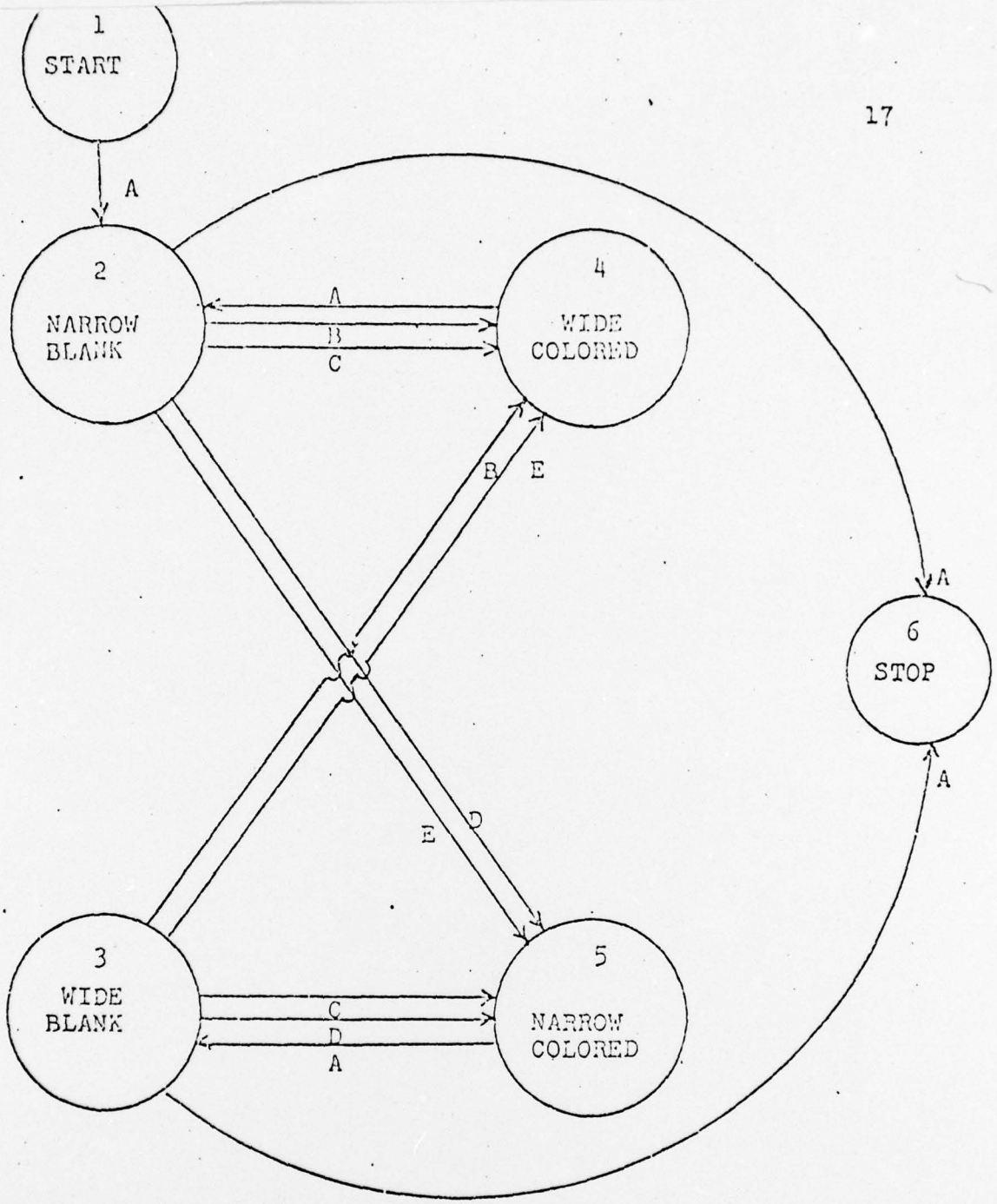
could find syntactic rules that would generate at least typical fragments of highly stylized pictures, say in the style of Mondrian.

The grammar tried was the following one. The words were five A,B,C,D, and E with the rules in the wiring diagram of Figure 3.3. Generate two sentences from this finite state automaton, code A,B,C,D,E into 0,1,2,3,4 with the color key

0 - white
1 - blue
2 - red
3 - yellow
4 - black

Use one sentence for horizontal effects, the other one vertically and add the key values modulo 5. Pictures are then obtained looking like the one in Figure 3.4. It can be questioned whether this is really Mondrian-like, but this is not the point here.

When the abduction machine was exposed to sentences convergence seemed to be much slower than for other, seemingly more difficult languages. It could look like this



THE MONDRIAN MACHINE

Figure 3.3

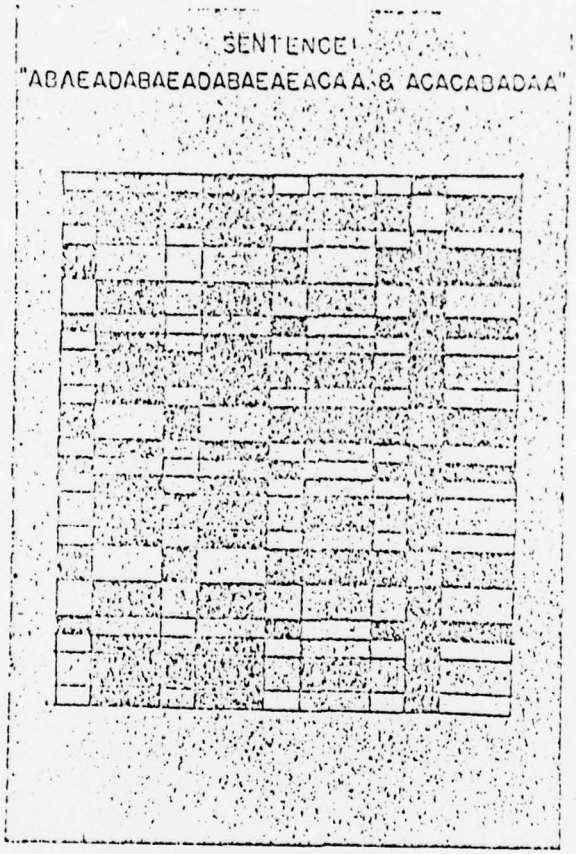


Figure 3.4

```
      TABULA
      LINGUA 20
ADADADAA
NEW VARIABLE A CREATED AT SENTENCE NO. 1 = ADADADAA
NEW VARIABLE AD CREATED AT SENTENCE NO. 1 = ADADADAA
AEAA
AEADADADADACADACADADAA
ACAEAEACACACAA
ADADADAEACADADADAEAEACADACAEACAEABADAEAEADACADAEADADACADAEACACACACAA
ACACACAEAA
AEAEAEACADACACAEAEACADAEACADADAEACADACACAA
ADACADADADADACAEACACAEAA
ADAEACACADAA
ACACAEADAEAEADACADACADADAEACAEADAEAEAEABADAA
ADADADACAA
ADADADAA
AEACADACADACADAA
ACADACACADADAEAEACACAA
AEADADACACAEAEACADACACADAEABACADAEAEACACACAEAA
ADACACACAA
AEAEADAEAEADAEAA
AEADACADAEAA
AA
NEW VARIABLE AA CREATED AT SENTENCE NO. 19 = AA
AEAEACAA
```

with only A,AD,AA as established variables after 20 iterations. Further iterations yielded no more variables. The sentences tend to be rather long, in one extreme case over 100 words long, but this should only marginally influence the computing time needed since most of it goes into processing short initial substrings.

The solution is quite simple. The grammar recovered is indeed weakly equivalent to the given one as can be seen by inspection of the wiring diagram in the automaton of Figure 3.5. The abduction machine has reconstructed a correct grammar although in a different, simpler form, equivalent but not equal to the one we started out from. It therefore seems that we could just as

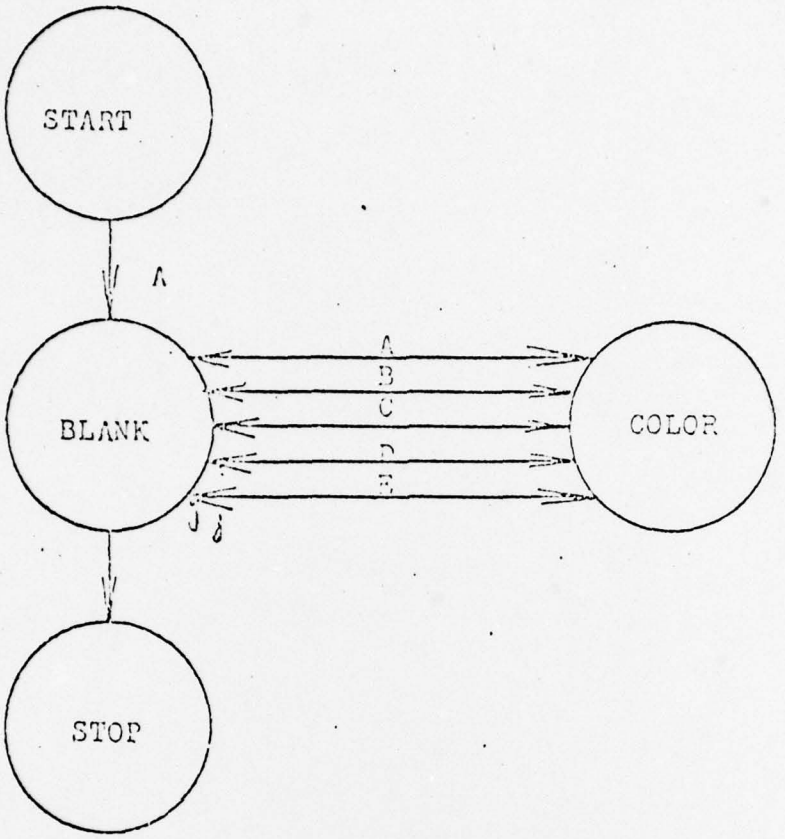


Figure 3.5

well have started out with the simpler form originally.

Not so. If we had done this it would not have been possible to differentiate the probabilities associated with the transitions between different colors and bands of different width. This would have meant that we could not have incorporated such stylistically significant elements.

This lesson teaches us that the notion of style that we have adopted here, is not so much a property of the grammar, as the usage of grammar. It is related to performance, not only competence.

If this point of view is accepted it means, as was the case for the Mondrian machine, that it may be necessary to increase the set of syntactic variables by other stylistic variables.

This raises the question of how to find adequate and "efficient" stylistic variables in situations of greater interest than this rather artificial example. It is clear that the abduction machine learns grammar but not style. Can one formalize the search for stylistic variables?

Appendix

To define a syntax-controlled probability mode the user should execute the function SETUP.

```
    ∇ SETUP
[1]   [illegible]←([illegible] 15)ρ0
[2]   '[illegible] 15 '
[3]   [illegible]←[illegible]
[4]   [illegible]'[illegible]'[ι[illegible]]
[5]   '[illegible] 5'
[6]   [illegible]←[illegible]
[7]   [illegible]←'12[illegible]'[ι[illegible]]
[8]   [illegible]←[illegible]ρ[illegible]
[9]   [illegible]←([illegible] 15)ρ0
[10]  [illegible]←([illegible] 15)ρ1
[11]  [illegible]←1
[12]  [illegible]←([illegible] 15 [illegible])ρ' '
[13] [illegible]:'[illegible] FOR VARIABLE NO. ';[illegible]
[14]  [illegible][[illegible]]←[illegible]
[15]  [illegible]←1
[16] [illegible]:'VARIABLE NO. ';[illegible];'[illegible] AT MOST 5'
[17]  [illegible]←,[illegible]
[18]  [illegible][[illegible];[illegible];ιρ[illegible]]←[illegible]
[19]  [illegible][[illegible];[illegible]]←ρ[illegible]
[20]  '[illegible] ?'
[21]  [illegible][[illegible];[illegible]]←[illegible]
[22]  [illegible]←[illegible]+1
[23]  →([illegible])/[illegible]
[24]  [illegible][[illegible];ι[illegible]]←((ι[illegible])∘.=[illegible])+.×[illegible][[illegible];ι[illegible]]
[25]  [illegible]←[illegible]+1
[26]  →([illegible])/[illegible]1
    ∇
```

It stores the information in the form of arrays LEXICON, RULES, LEN, CUM. In order to save space the user has been restricted to at most 15 words, 9 variables, and 5 rewriting rules for each variable. If this is not enough the statements [1], [9], [10], and [12] should be changed accordingly. The initial variable is the one labelled 1.

After calling SETUP execution of the function REWRITE will generate one grammatical string named STRING according to the model STRING is a lexical vector, but if one needs a numerical vector this is easily obtained by the statement

LEXICON ⍳ STRING

resulting in a vector of the same length but with each word replaced by the number of it in the LEXICON.

```
     ∇ REWRITE
[1]    STRING←,'1'
[2]   RL1:L←⍴STRING
[3]    SUB←,STRING∘.=VARIABLES
[4]    →(0≠∨/SUB)/RL2
[5]    →0
[6]   RL2:INDEX←SUB⍳1
[7]    VAR←STRING[INDEX]
[8]    VARINDEX←+/(VAR=VARIABLES)/⍳NV
[9]    NUM←1++/CUMVARINDEX;]<1E¯6×?1000000
[10]   SUBSTRING←RULES[VARINDEX;NUM;⍳LEN[VARINDEX;NUM]]
[11]   STRING←STRING[⍳INDEX-1],SUBSTRING,STRING[INDEX+⍳L-INDEX]
[12]   →RL1
     ∇
```

The function ACCEPT takes an input string SEN as an alphabetic vector. The result Z is 1 or 0 according to whether the string SEN is grammatical or not.

```
          ∇ Z←ACCEPT SEN
[1]       SEN←,SEN
[2]       ST←1
[3]      AL1:ST←MATRIX[ST;LEXICON⍳SEN[1]]
[4]       SEN←1↓SEN
[5]       →(ST=0)/AL2
[6]       →(0≠⍴SEN)/AL1
[7]       Z←0
[8]       →0
[9]      AL2:→(0=⍴SEN)/AL3
[10]      Z←0
[11]      →0
[12]     AL3:Z←1
```

It uses an array MATRIX whose rows are the numbers (in LEXICON) of the words and whose columns are the states 1,2,3,... if necessary increased with additional states as described in the text. The entries are the next state where 0 stands for the final state. The automaton in Figure 2.1 for example would have

$$\text{MATRIX} = \begin{Bmatrix} 1 & 2 & 4 \\ 4 & 3 & 0 \\ 1 & 0 & 0 \\ 4 & 4 & 4 \end{Bmatrix}$$

The function TABULA sets up one single equivalence class and some selected variables. The matrix CLASS will have NCOL number of columns = n_c.

```
     ∇ TABULA
[1]    CU←0,((⍳D)∘.≥⍳D)+.×MM*⍳D
[2]    NCOL←CU[D+1]
[3]    LM←1
[4]    CLASS←(1,NCOL)⍴1
[5]    POZ←1
     ∇
```

The function LINGUA has one argument MORE = number of additional sentences to be generated and presented to the listener. It uses the subroutines CODE which takes the number of the substring and produces the substring and DECODE which is the inverse function, both for substring in the form of numerical vectors.

```
     ∇ CX←CODE X
[1]    DEPTH←+/X>CU
[2]    CX←1+(DEPTHρNV)⊤¯1+X-CU[DEPTH]
     ∇

     ∇ X←DECODE X
[1]    X←1+CU[TAKE]+(TAKEρNV)⊥X-1
     ∇
```

LINGUA prints out each sentence heard and calls the comparison function TEST.

```
     ∇ LINGUA MORE
[1]    T←1
[2]   LL1:REWRITE
[3]    STRING
[4]    SENT←LEXICON⍳STRING
[5]    TAKE←1
[6]   LL2:BEGIN←,SENT[⍳TAKE]
[7]    V←DECODE BEGIN
[8]    SET←CLASS[CLASS[;V]/⍳LV;]
[9]    SET←SET[1;]/⍳NCOL
[10]   TESTV←SET[?(ρSET)[1]]
[11]   TEST
[12]   TAKE←TAKE+1
[13]   →(TAKE≤D⌊ρSENT)/LL2
[14]   T←T+1
[15]   TAKE←1
[16]   TOT←TOT+1
[17]   →(T≤MORE)/LL1
     ∇
```

TEST prints out each decision to introduce a new class, the substring corresponding to it, the sentence number when it occurred, and the sentence itself.

Afterwards typing CLASS will print out the CLASS array which tells us how the classes that have been established up till now are composed from substrings. This will enable us to easily reconstruct the corresponding rewriting rules and the

wiring diagram of the finite state automaton that generates the language as it has been learnt up till now.

```
     ∇ TEST
[1]    U←CODE TESTV
[2]    YES←ACCEPT LEXICON[U,TAKE↓SENT]
[3]    →YES/0
[4]    COMP←,(CLASS[;V]=0)/⍳LV
[5]   TL1:→(0=⍴COMP)/TL2
[6]    ROW←(CLASS[COMP[1];])/⍳NCOL
[7]    ROW←ROW[?⍴ROW]
[8]    TESTW←CODE ROW[1]
[9]    →(~ACCEPT LEXICON[TESTW,TAKE↓SENT])/TL3
[10]   CLASS[(CLASS[;V]=1)/⍳LV;V]←0
[11]   CLASS[COMP[1];V]←1
[12]   →0
[13]  TL2:CLASS[(CLASS[;V]=1)/⍳LV;V]←0
[14]   LV←LV+1
[15]   'NEW VARIABLE ',LEXICON[BEGIN],' CREATED AT SENTENCE NO. ';TOT;'= ',STRING
[16]   CLASS←(LV,NCOL)⍴(.,CLASS),V=⍳NCOL
[17]   →0
[18]  TL3:COMP←1+COMP
[19]   →TL1
     ∇
```

Reference

U. Grenander (1966): Can we look inside an unreliable automaton? Festschrift for J. Neyman. John Wiley & Sons, New York.